PRÁCTICA 2:

SISTEMA DE COCHES

- Juan Manuel Valenzuela González – amcgil@uma.es
- Artur Vargas Carrión – arturvargas797@uma.es
- Jorge Repullo Serrano – jorgers4@uma.es
- Eduardo González Bautista – edugb@uma.es
- Rubén Oliva Zamora – rubenoliva@uma.es
- David Muñoz del Valle – davidmunvalle@uma.es
- Alejandro Jiménez González – alejg411@uma.es

Modelado y Diseño del Software

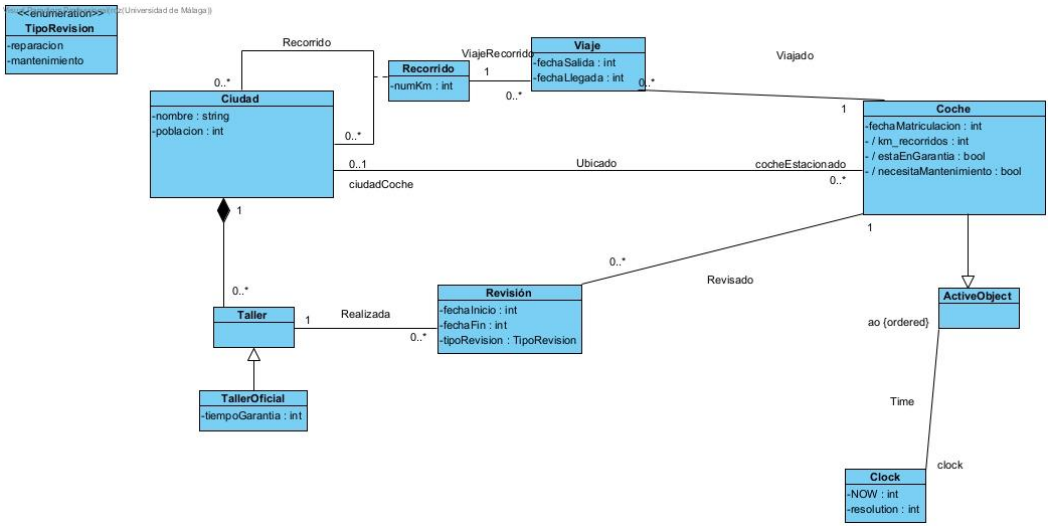
GR1-3 | Ingeniería del Software A | Universidad de Málaga

ÍNDICE DE CONTENIDO

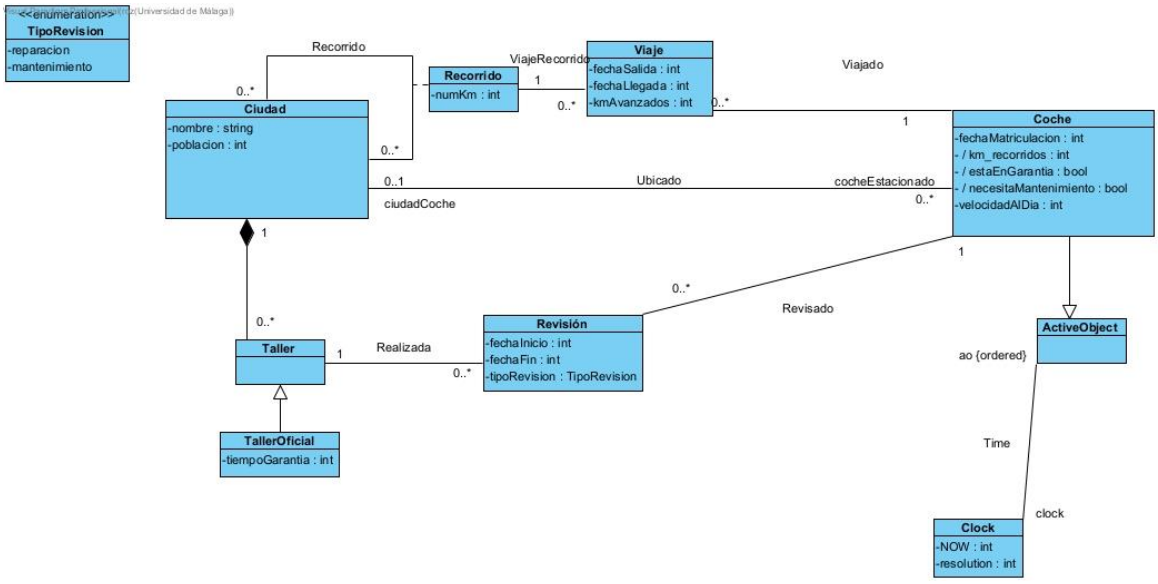
1. Visual Paradigm

Se ha utilizado Visual Paradigm (VP) para realizar el **modelado conceptual base** (primera aproximación) del sistema de coches. Este entorno nos ha permitido crear un primer diseño del sistema, identificando las entidades principales y las relaciones entre ellas, lo que nos ha permitido iterar y refinar el modelo hasta llegar a la versión final. VP nos ha sido de gran utilidad para visualizar y entender de forma clara la estructura del sistema antes de pasar a las fases de validación y verificación.

El sistema que hemos modelado recopila información sobre coche, taller, ciudad y viaje gestionando aspectos clave como los viajes que realiza el coche, qué recorrido debe hacer y entre qué ciudades es, si está siendo reparado en un taller y en qué ciudad se encuentran tanto el taller como el coche. Este modelo tiene como objetivo garantizar una representación fiel de las interacciones y restricciones que rigen el sistema de coche.

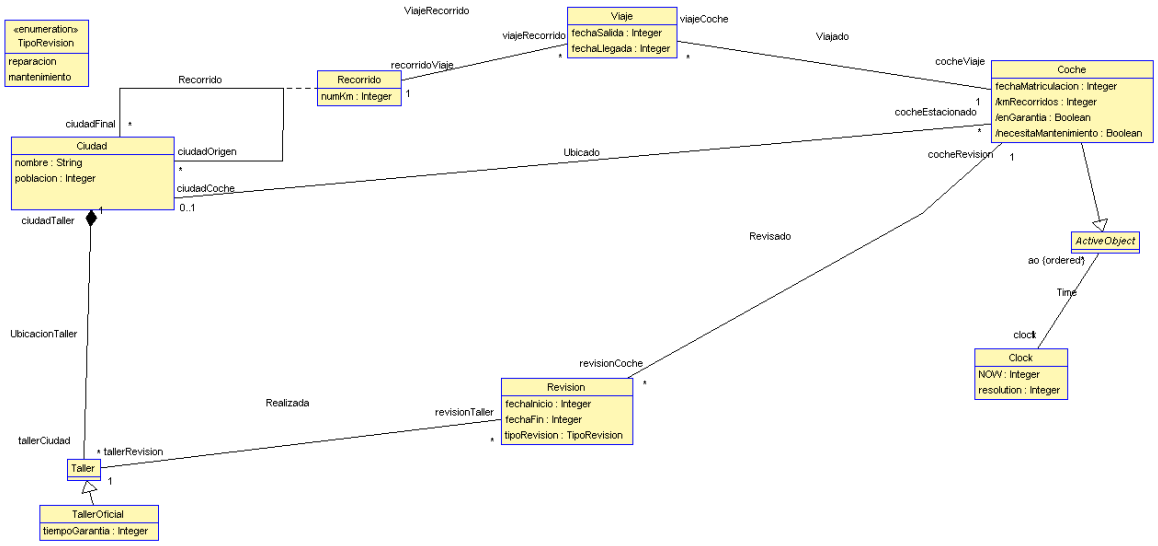


1 Diagrama de clases del apartado A del sistema de coches en Visual Paradigm

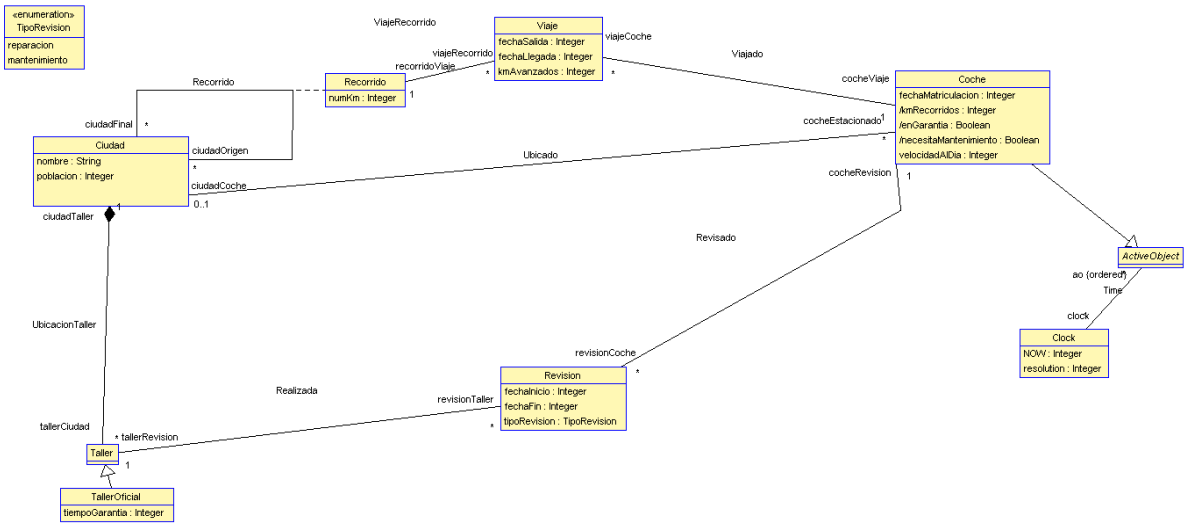


2 Diagrama de clases del apartado B del sistema de coches en Visual Paradigm

2. USE (UML-based Specification Environment)



3 Diagrama de clases del apartado A del sistema de coches en USE



4 Diagrama de clases del apartado B del sistema de coches en USE

2.1. Aclaraciones sobre clases

Inicialmente, en el diagrama teníamos la clase **`Revision`** como una **clase de asociación** entre **`Coche`** y **`Taller`**. Sin embargo, esto presentaba un problema: cuando un coche iba varias veces al mismo taller, la información de cada revisión se sobrescribía en lugar de crear una nueva instancia de **`Revision`** para cada visita. Esto se debía a que una clase de asociación permite una única instancia entre los mismos objetos relacionados. Para resolver este inconveniente, decidimos modelar **`Revision`** como una **clase intermedia**. De esta manera, cada revisión de un coche en un taller se representa como una instancia separada, lo cual permite almacenar múltiples revisiones del mismo coche en el mismo taller.

De igual forma, optamos por modelar **`Viaje`** como clase intermedia para poder registrar múltiples instancias de **`Viaje`** para el mismo **`Coche`** en el mismo **`Recorrido`**. Este enfoque permite llevar un registro detallado de cada evento y evitando la sobreescritura de información.

Para representar la distinción entre talleres oficiales y no oficiales, se ha diseñado una estructura en la que la clase **`Taller`** actúa como clase base, de la cual hereda la clase **`TallerOficial`**. De esta forma, un objeto que pertenezca únicamente a la clase **`Taller`** se interpreta como un taller no oficial, mientras que aquellos que son instancias de **`TallerOficial`** representan talleres oficiales.

2.2. Aclaraciones sobre relaciones y atributos

2.2.1. Relaciones

La relación de **`Ciudad` con `Ciudad` tiene multiplicidad 0..*** porque de una ciudad se puede ir a otras muchas ciudades, no solo a una. Hicimos este cambio porque al hacer pruebas con los soils descubrimos que **se violaba una restricción de multiplicidad**, que se debía a que teníamos la relación mencionada como 1 a 1.

Para identificar la **ubicación** de un coche en un momento dado, hemos establecido una relación entre las clases **`Coche`** y **`Ciudad`**. Esta relación tiene una **multiplicidad de 0 o 1**. Esto significa que un coche puede no estar asociado a ninguna ciudad (multiplicidad 0) cuando está de viaje, indicando que no se encuentra en una ubicación específica. Sin embargo, cuando el coche no está de viaje, la relación tiene una multiplicidad de 1, señalando que se encuentra en una ciudad determinada.

Para modelar adecuadamente el **historial** de un **coche**, hemos definido relaciones de multiplicidad 0..* entre **`Coche`** y las clases **`Revisión`** y **`Viaje`**. Esto significa que **un coche puede estar relacionado con múltiples instancias de Revisión y Viaje**, reflejando su historial completo de revisiones en distintos talleres y de los viajes que ha realizado.

Para saber si un **coche** se encuentra actualmente en un **taller** o en un **viaje**, utilizamos las propiedades **fechaFin** en **`Revisión`** y **fechaLlegada** en **`Viaje`**. Si **fechaFin** de una **revisión** es **indefinida**, esto indica que el coche está actualmente en ese taller. De manera similar, si **fechaLlegada** en un **viaje** es **indefinida**, se interpreta que el coche se encuentra en ese viaje en ese momento. Esta estructura permite registrar el historial completo de revisiones y viajes mientras se identifica el estado actual del coche según el valor de estas fechas.

2.2.2. Atributos derivados

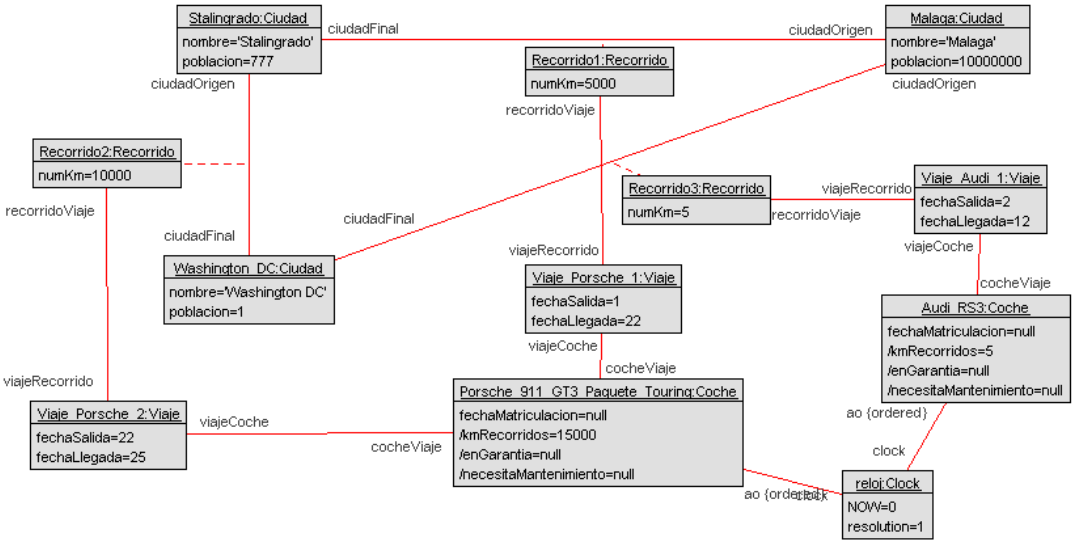
Para comprobar la eficacia de los atributos derivados hemos realizado un archivo soil por atributo derive implementado.

- **Coche → kmRecorridos**

Calcula los kilómetros recorridos, sumando los kilómetros de todos los viajes que el coche ha terminado.

```
    kmRecorridos : Integer derive :
      self.viajeCoche->select(v | v.fechaLlegada.isDefined())->collect(v |
v.recorridoViaje.numKm)->sum()
```

Abajo se muestra el diagrama de objetos que calcula este atributo en base a los viajes realizados:



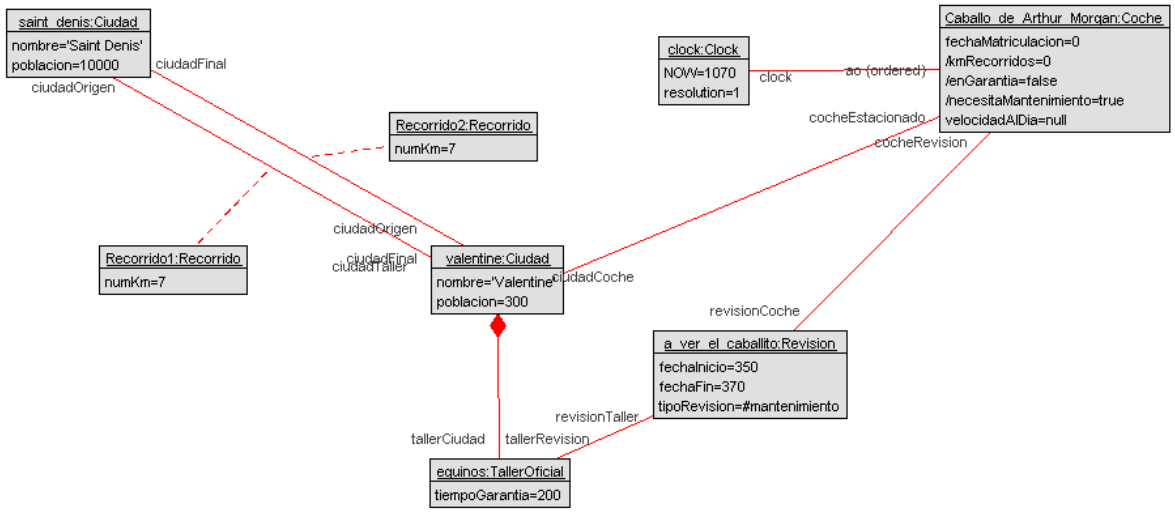
En el apartado b se calcula de otra forma. Esto es explicado en el punto 2.4.1 y probado en el apartado c, como se muestra en el punto 2.4.3.3 del documento.

- **Coche → enGarantia**

Comprueba si el coche está en garantía. Para ello, está en garantía si no han pasado más de 4 años desde que se matriculó o si relazó alguna revisión en un taller oficial y le queda garantía.

```
enGarantia : Boolean derive :
      ((self.clock.NOW - self.fechaMatriculacion) <= 4 * 100) or -- No han
pasado más de 4 años desde la matriculacion
      -- Hay una revisión en un taller oficial y aun le queda garantia
      self.revisionCoche->exists(r |
r.tallerRevision.oclIsTypeOf(TallerOficial) and (self.clock.NOW -
r.fechaFin <= r.tallerRevision.oclAsType(TallerOficial).tiempoGarantia))
```

Abajo se muestra el diagrama de objetos que en el atributo está a false, ya que, aunque el coche haya realizado una revisión, ya ha pasado el tiempo de garantía:

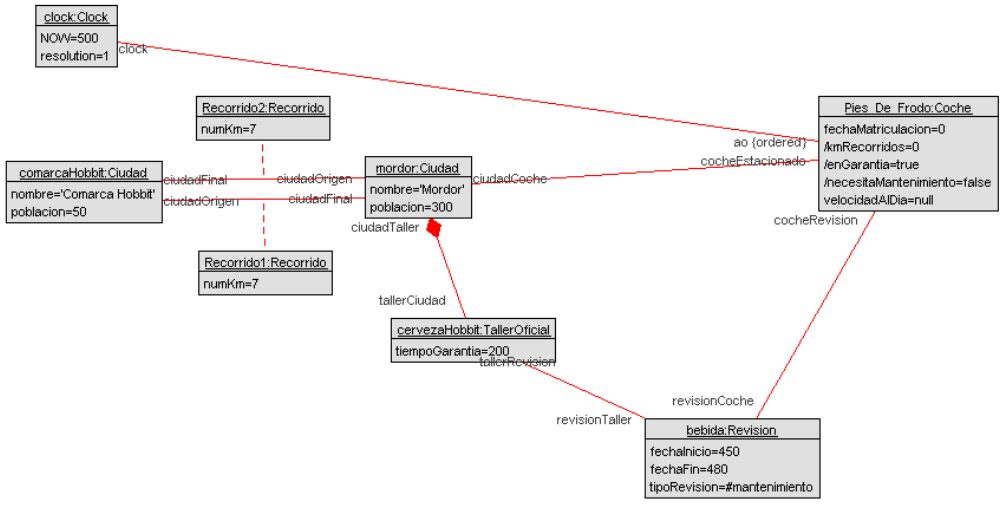


- **`Coche → necesitaMantenimiento`**

Comprueba si el coche necesita mantenimiento. Para ello, no necesita mantenimiento si no han pasado más de 4 años desde que se matriculó o si han pasado más de 4 años desde que se matriculó y hace menos de un año que realizó una revisión de mantenimiento.

```
necesitaMantenimiento : Boolean derive :
        -- Saco la fecha del último mantenimiento
        let fechaUltimoMantenimiento : Integer = (self.revisionCoche->select(r
| r.tipoRevision = TipoRevision::mantenimiento)->sortedBy(r | r.fechaFin)-
>last().fechaFin) in

        ((self.clock.NOW - self.fechaMatriculacion) > 4 * 100) and -- Han
pasado más de 4 años desde la matriculacion
        ((self.revisionCoche->isEmpty()) or -- Si no tiene revisiones, le toca
revisión
          (self.clock.NOW - fechaUltimoMantenimiento >=  100)) -- Si las
tiene, le toca si pasó 1 año desde la última de mantenimiento
```

Abajo se muestra el diagrama de objetos que el atributo está a false, pues han pasado 4 años desde que se matriculó, pero el coche ha realizado una revisión de mantenimiento y no ha pasado más de un año desde esta (NOW está a 500 y la fecha de fin de revisión está en 480):



2.3. Invariantes

Para comprobar el correcto funcionamiento de las invariantes hemos realizado un archivo soil para cada uno de ellos. En estos archivos soil, se comprenden los distintos escenarios posibles para que se cumpla o viole cada invariante, indicados con comentarios. Como no queríamos sobrecargar el documento con código, hemos decidido no poner el código de cada soil aquí. Por favor, para más detalle, se ruega que se consulte cada archivo soil.

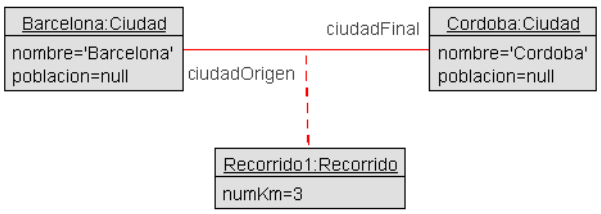
2.3.1. Explicación de invariantes

- **`Recorrido → DistanciaEntreCiudadesAlMenosCinco`**

Asegura que la distancia entre dos ciudades en cualquier recorrido sea, como mínimo, de cinco kilómetros. Esto garantiza que no se registren distancias menores, manteniendo la consistencia en las distancias mínimas del sistema.

```
inv DistanciaEntreCiudadesAlMenosCinco :
    self.numKm >= 5
```

Abajo se muestra el diagrama de objetos que violaría este invariante. La distancia entra las dos ciudades es de 3 km, menos de 5 km:

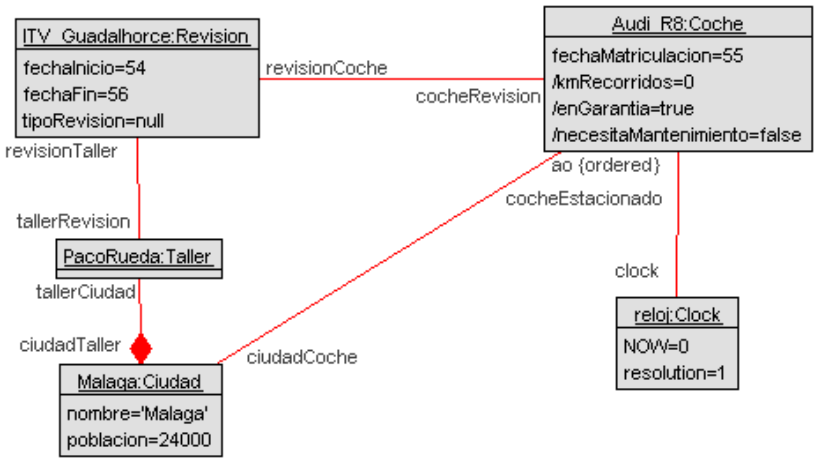


- **`Revision → RevisionDespuesDeMatriculacion:`**

Asegura que ninguna revisión de un coche ocurra antes de su matriculación, es decir, que la fecha de matriculación siempre sea anterior a la fecha de inicio de la revisión.

```
inv RevisionDespuesDeMatriculacion :
      self.cocheRevision.fechaMatriculacion < self.fechaInicio
```

Aquí el diagrama de objetos que incumple la invariante. El audiR8 fue matriculado el día 55, mientras que la revisión comenzó el día 54:

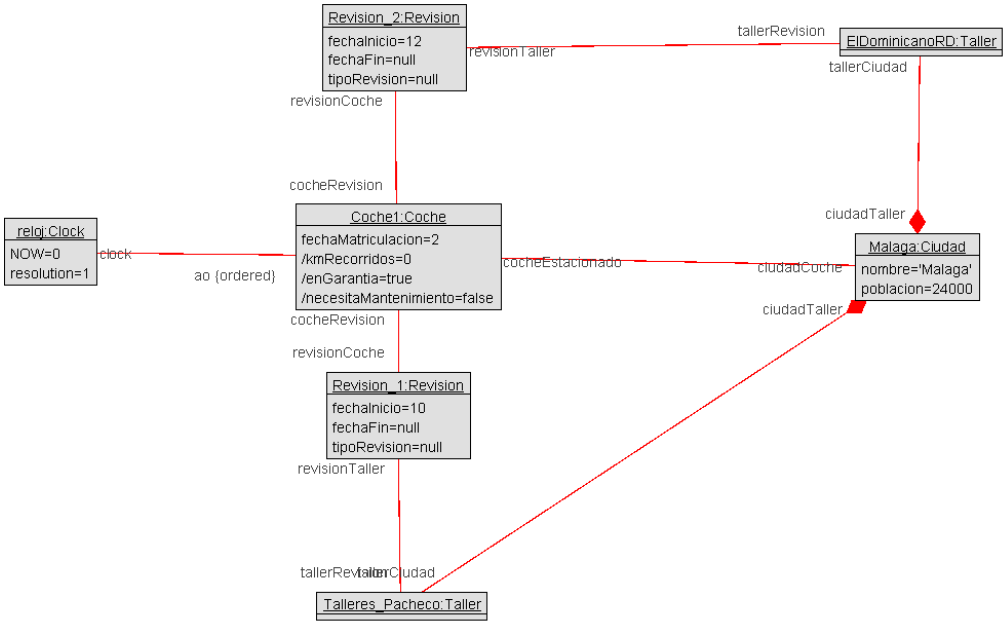


- **Revision → UnaRevisionALaVez:**

Asegura de que no ocurran dos revisiones a la vez. Es decir, que no haya dos revisiones con "fechaFin" nula.

```
inv UnaRevisionALaVez :
    self.revisionCoche->select(r | r.fechaFin.isUndefined())->size() <= 1
```

Aquí el diagrama de objetos que incumple la invariante. El Coche1 está en dos revisiones a la vez:

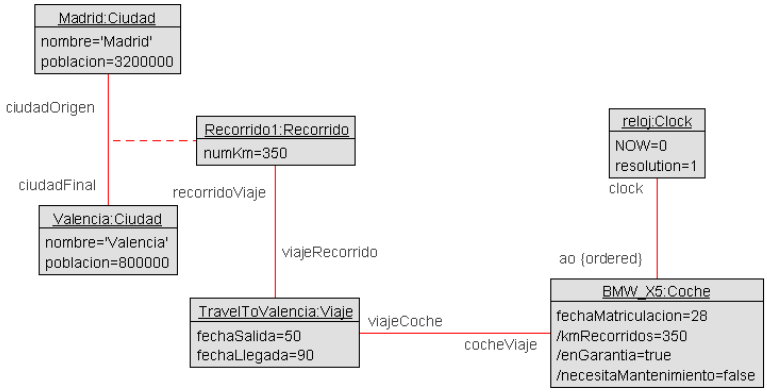


- **Viaje → ViajeEnCurso**:

Establece que un viaje en curso (con la relación **Ubicado** indefinida) no debe tener una fecha de llegada registrada, indicando que el coche aún está en movimiento.

```
inv ViajeEnCurso :
      self.cocheViaje.ciudadCoche.isUndefined() implies
self.fechaLlegada.isUndefined()
```

Aquí nos encontramos el diagrama de objetos que no cumple el invariante. El coche se encuentra en un viaje que no ha terminado (todavía no está ubicado en una ciudad), por tanto, la fecha de llegada no debería estar determinada. Cabe destacar que este diagrama de objetos también incumple los invariantes CocheTerminaEnCiudad (el viaje ha terminado, pero no hay relación Ubicado) y CocheEnViajeOCiudad (no se encuentra ni viajando ni en una ciudad, ya que la fecha de llegada está definida y no hay relación ubicado).

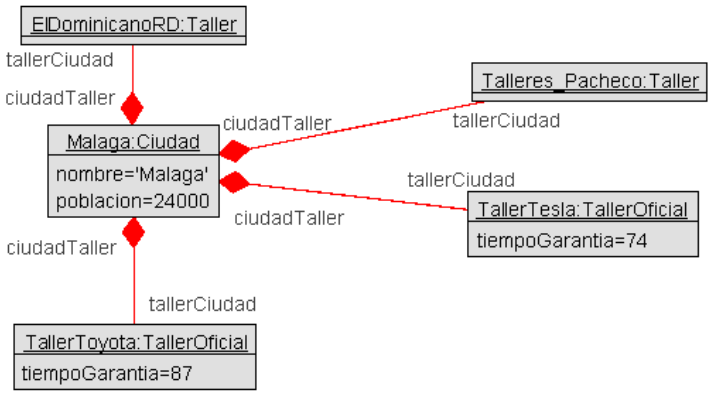


- **Ciudad → UnSoloTallerOficial:**

Comprueba que en cada ciudad haya solo un único taller oficial.

```
inv UnSoloTallerOficial :
      self.tallerCiudad->select(t | t.oclIsTypeOf(TallerOficial))->size() <= 1
```

Abajo se encuentra el diagrama de objetos que incumple el invariante. En la ciudad de Málaga únicamente puede existir un taller oficial, mientras que en este diagrama de objetos podemos apreciar 2, TallerToyota y TallerTesla.

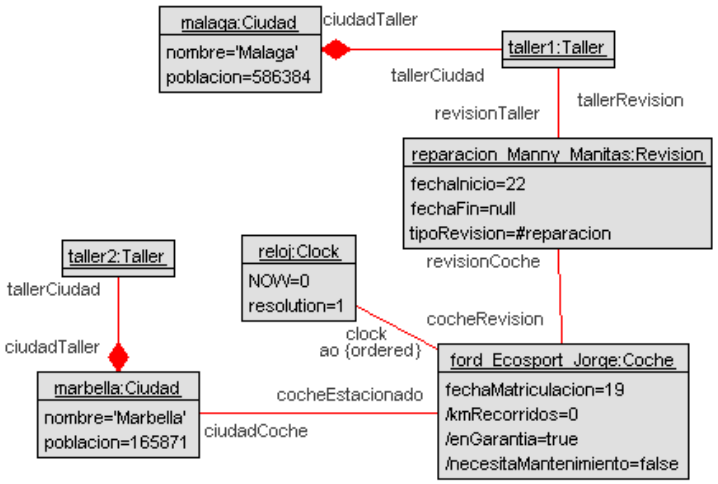


- **Coche → CiudadCocheEnRevision_Igual_CiudadTaller:**

Comprueba que, si el coche está en revisión, este se encuentre en la misma ciudad donde se ubica el taller que está realizando esta revisión.

```
inv CiudadCocheEnRevision_Igual_CiudadTaller :
      self.revisionCoche->isEmpty() or  -- no tiene revisiones
      self.revisionCoche->exists(r | r.fechaFin.isUndefined() implies
self.ciudadCoche = r.tallerRevision.ciudadTaller)
```

Aquí muestro el diagrama de objetos que incumple el invariante. El Ford EcoSport de Jorge aparece que está estacionado en Marbella, sin embargo, lo están reparando en el taller de Manny Manitas que está ubicado en la ciudad de Málaga.

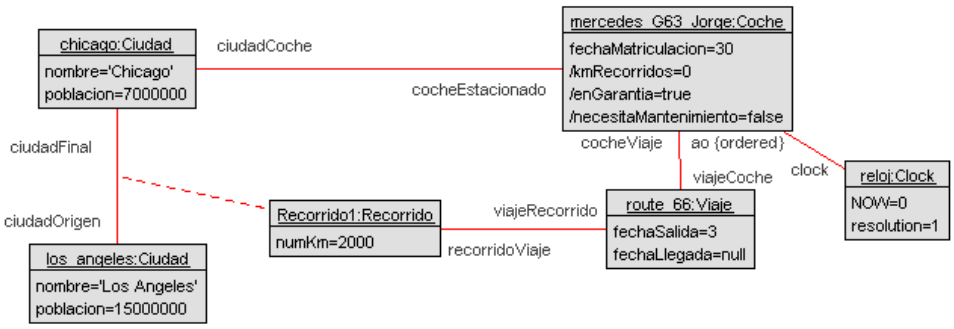


- **Coche → CocheEnViajeOCiudad:**

Comprueba que el coche, en caso de que se encuentre en un viaje, no estará ubicado en una ciudad, o bien que, si tiene una ciudad asignada, ningún viaje tendrá la fecha de llegada sin definir, o no existirá ningún viaje anterior.

```
inv CocheEnViajeOCiudad :
      (self.viajeCoche->isEmpty() and self.ciudadCoche.isDefined()) or  -- si
nunca ha hecho un viaje, está en una ciudad
      (self.viajeCoche->exists(v | v.fechaLlegada.isUndefined()) and
self.ciudadCoche.isUndefined()) or -- si está haciendo un viaje, no está en
una ciudad
      (self.ciudadCoche.isDefined() and not self.viajeCoche->exists(v |
v.fechaLlegada.isUndefined())) -- si está en una ciudad, no está haciendo un
viaje
```

Abajo se muestra el diagrama de objetos que incumple el invariante. El Mercedes G63 de Jorge se encuentra en la ciudad de Chicago, pero a la vez está en un viaje.

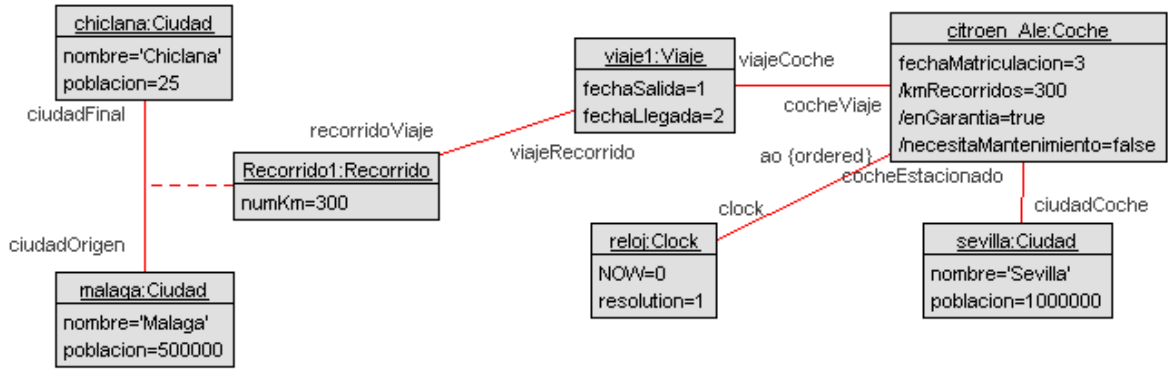


- **Coche → CocheTerminaEnCiudad:**

Comprueba si el coche ha completado al menos un viaje y no se encuentra viajando, por lo que debe encontrarse en la ciudad a la que llegó en su último viaje.

```
inv CocheTerminaEnCiudad :
      (self.viajeCoche->isEmpty()) or   -- Tiene que dar correcto si ha
completado 0 viajes
      (not self.viajeCoche->exists(v | v.fechaLlegada.isUndefined())
implies -- si el coche no está en un viaje
      (self.ciudadCoche = self.viajeCoche->sortedBy(v | v.fechaLlegada)-
>last().recorridoViaje.ciudadFinal)) -- la ciudad en la que está ahora es
la ciudad a la que ha llegado en el último viaje
```

Este diagrama de objetos muestra cómo se incumple este invariante. El Citroën de Alejandro ha terminado el viaje Chiclana-Málaga, pero no se encuentra en ninguna de estas 2 ciudades, se encuentra en Sevilla.

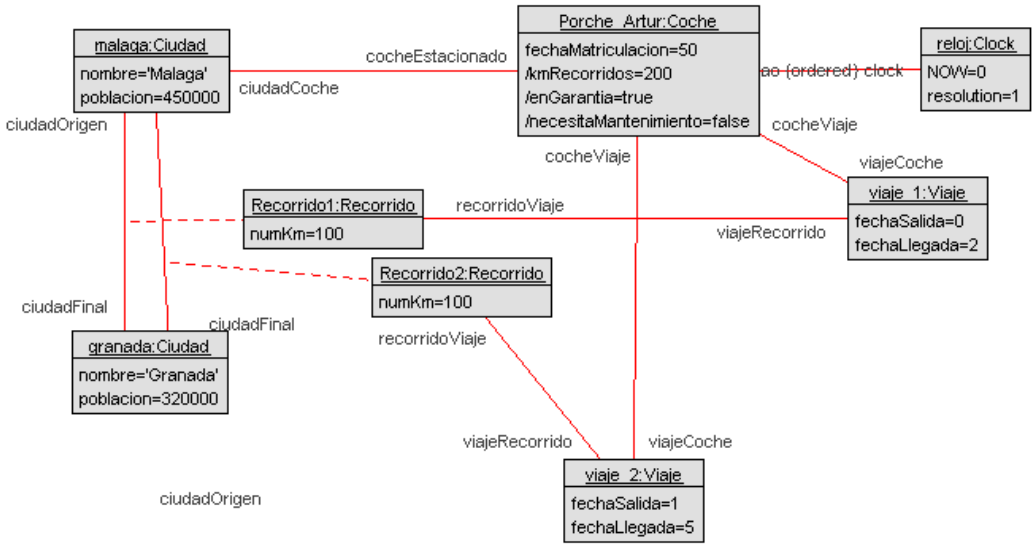


- **Coche → ViajesNoSeSolapan:**

Los coches no pueden realizar dos viajes en el mismo tiempo, por tanto, los viajes siempre deben ocurrir uno después del otro.

```
inv ViajesNoSeSolapan :
      let viajesOrdenados : OrderedSet(Viaje) = self.viajeCoche->sortedBy(v
| v.fechaSalida) in -- Saco los viajes ordenados
       (self.viajeCoche -> size() >= 2) implies -- Si tengo más de 2
       (self.viajeCoche-> forAll(v1,v2 | viajesOrdenados->indexOf(v1) =
                                  viajesOrdenados->indexOf(v2) - 1 implies
((v1.fechaLlegada <= v2.fechaSalida)))) -- basicamente,
viaje[i].fechaLlegada < viaje[j].fechaSalida | i < j
```

Este diagrama de objetos incumple la invariante, debido a que viaje_2 comienza durante el viaje_1.

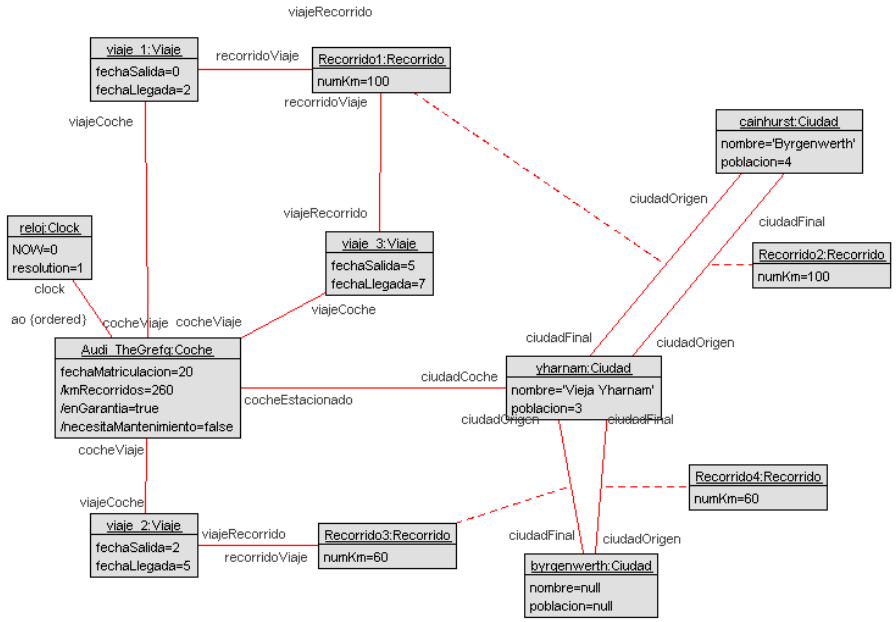


- **Coche → CiudadesCoherentes**

Comprueba si las ciudades de origen y destino de los viajes deben ser coherentes. Es decir, si un coche realiza un viaje desde la ciudad A hasta la ciudad B, el próximo viaje debe partir desde la ciudad B.

```
inv CiudadesCoherentes :
      let viajesOrdenados : OrderedSet(Viaje) = self.viajeCoche->sortedBy(v
| v.fechaSalida) in -- Saco los viajes ordenados
        (self.viajeCoche -> size() >= 2) implies -- Si tengo más de 2
        (self.viajeCoche-> forAll(v1,v2 | viajesOrdenados->indexOf(v1) =
                                  viajesOrdenados->indexOf(v2) - 1 implies
(v1.recorridoViaje.ciudadFinal = v2.recorridoViaje.ciudadOrigen)))
```

El diagrama de objetos siguiente no cumple la invariante debido a que The_Grefg con su Audi termina el primer viaje en Cainhurst mientras que el segundo viaje que realiza comienza en Yharnam.



Además de los invariantes que nos pide implementar el enunciado, hemos introducido algunos **invariantes implícitos** al funcionamiento del sistema. Estos no se mencionan explícitamente, pero su adición no debe cambiar el funcionamiento del sistema; al contrario, debe hacerlo más robusto. Dichas invariantes implícitos son:

- **Clock → RelojEsUnico:**

Este invariante comprueba que únicamente haya un reloj en el sistema.

```
inv RelojEsUnico :
    Clock.allInstances()->size()=1
```

- **Revisión → FechasRevisionValidas**

El invariante comprueba que todas las fechas sean positivas. También que la fecha de llegada sea anterior a la fecha de salida y la de llegada siempre esté definida.

```
inv FechasRevisionValidas :
    self.fechaInicio.isDefined() and self.fechaInicio >= 0 and
    (self.fechaFin.isDefined() implies (self.fechaInicio <=
self.fechaFin and self.fechaFin>=0))
```

- **Viaje → FechasViajeValidas**

El invariante "FechasViajeValidas" comprueba que la fecha de llegada siempre sea posterior a la de salida, que las fechas siempre sean positivas y que las fechas de estos viajes sean posteriores a la fecha de matriculación del coche que lo realiza.

```
inv FechasViajeValidas :
    self.fechaSalida.isDefined() and self.fechaSalida >= 0 and
    (self.fechaLlegada.isDefined() implies (self.fechaSalida <=
self.fechaLlegada and self.fechaLlegada >= 0)) and
    self.cocheViaje.fechaMatriculacion <= self.fechaSalida
```

- **Coche → FechaMatriculacionValida**

Esta invariante comprueba simplemente que la fecha de matriculación de un Coche sea mayor que 0.

```
inv FechaMatriculacionValida :
    self.fechaMatriculacion >= 0
```

2.4. Sobre el apartado B y C

2.4.1. Cambios respecto al apartado A

El modelo del que partimos es el mismo, salvo por algunas pequeñas diferencias. Se ha añadido el atributo **velocidadAlDia : Integer** en la clase **Coche**, que guarda la velocidad a la que siempre irá un coche siempre y cuando se encuentre en un viaje. Este atributo no variará, ya que no se pide que su comportamiento sea "dinámico" en el enunciado del ejercicio.

También se ha añadido el atributo **kmAvanzados : Integer init : 0** en la clase **Viaje**, inicializado a 0 ya que, por definición, cuando un viaje comienza aún no se ha recorrido kilometraje del mismo. Este atributo irá variando cada vez que el coche avance con la operación **avanzar()**, que suma al atributo arriba mencionado el kilometraje realizado en un día, es decir, se le suma el atributo **velocidadAlDia**.

Además, el atributo **kmRecorridos** de la clase Coche ahora se calcula de otra forma. En lugar de ser el resultado de sumar los kilómetros una vez el viaje ha finalizado, ahora es el resultado de la suma de los **kmAvanzados** en cada viaje, por lo que los **kmRecorridos** del vehículo se van actualizando "en tiempo real".

```
kmRecorridos : Integer derive :
 self.viajeCoche->collect(v | v.kmAvanzados)->sum()
```

2.4.2. Aclaraciones sobre operaciones

Aparte de la operación **tick()** del **Clock** y la consecuente operación **action()** del vehículo, hemos tenido que implementar las operaciones **comenzarViaje(r : Recorrido)** y **avanzar()**.

La primera tiene tanto precondición como postcondición, pero **la segunda solo tiene postcondición**. Esto se debe a que inicialmente habíamos pensado que la precondición para avanzar era que el coche estuviese en un viaje, pero tal y como se ha implementado, esto debe ser definido como un if en **`action()`**, ya que si se pone como precondición, **cada vez que ocurra un tick de reloj y haya un coche que no esté de viaje, dará un fallo en la precondición** y el código no se podrá seguir ejecutando.

En cuanto al funcionamiento de las operaciones, **`comenzarViaje()`** crea un viaje y asigna las relaciones pertinentes, que son **`Viajado`** y **`ViajeRecorrido`**. También borra la relación **`Ubicado`**, para denotar que el vehículo ya no se encuentra en una ciudad, sino que se encuentra viajando.

Por otro lado, **`avanzar`**() simplemente selecciona el viaje que está haciendo el coche y le suma la **`velocidadAlDia`** hasta completar el recorrido del viaje. Cabe destacar que **siempre que se llama a `avanzar`() hay un viaje en curso**, ya que, como hemos mencionado anteriormente, **`avanzar`**() se llama desde **`action`**(), y **se llama solo si cumple la condición del if, que es precisamente que haya un viaje en curso**.

2.4.3. Aclaraciones generales

Cabe destacar que **todos los soils que se han implementado han sido probados con el comando check** en la línea de comandos de USE, para así comprobar que no se ha violado ninguna restricción de multiplicidad y/o estructura.

Con respecto al apartado C, para comprobar el correcto funcionamiento de la simulación se debe abrir apartado_c.soil e insertar los comandos indicados en apartado_c.usecmd. Se pueden cambiar los `!clock.run(n)` por `!clock.tick()` si se desea comprobar poco a poco el correcto funcionamiento del sistema. El contenido de apartado_c.soil y apartado_c.usecmd son los siguientes.

2.4.3.1. apartado_c.soil

```
reset

-- Clock
!new Clock('clock')


-- Ciudades
!new Ciudad('Malaga')
!Malaga.nombre := 'Malaga'
!Malaga.poblacion := 2000

!new Ciudad('Sevilla')
!Sevilla.nombre := 'Sevilla'
!Sevilla.poblacion := 777

!new Ciudad('Granada')
!Granada.nombre := 'Granada'
```

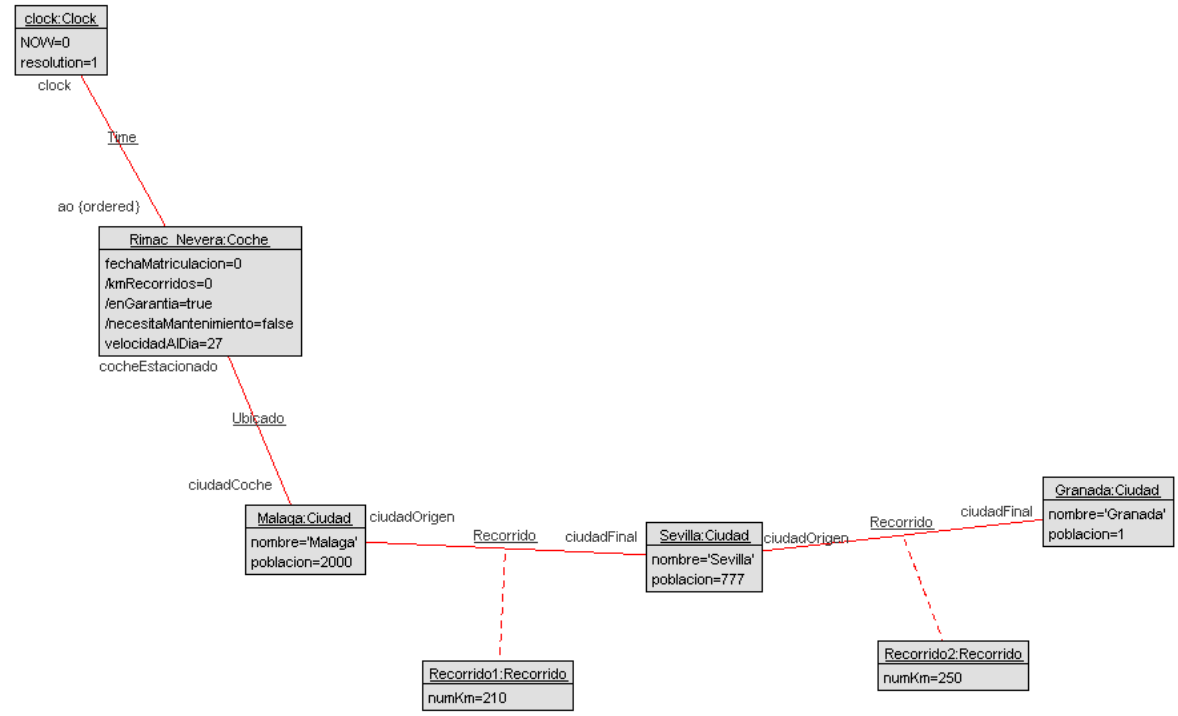
```
!Granada.poblacion := 1

-- Recorridos
!insert(Malaga,Sevilla) into Recorrido
!Recorrido1.numKm := 210

!insert(Sevilla,Granada) into Recorrido
!Recorrido2.numKm := 250

-- Coche
!new Coche('Rimac_Nevera')
!Rimac_Nevera.fechaMatriculacion := 0
!Rimac_Nevera.velocidadAlDia := 27

-- Necesario para que el coche se pueda comunicar con el clock
!insert(clock, Rimac_Nevera) into Time

-- El coche comienza en Málaga
!insert(Rimac_Nevera, Malaga) into Ubicado
```

2.4.3.2. apartado_c.usecmd

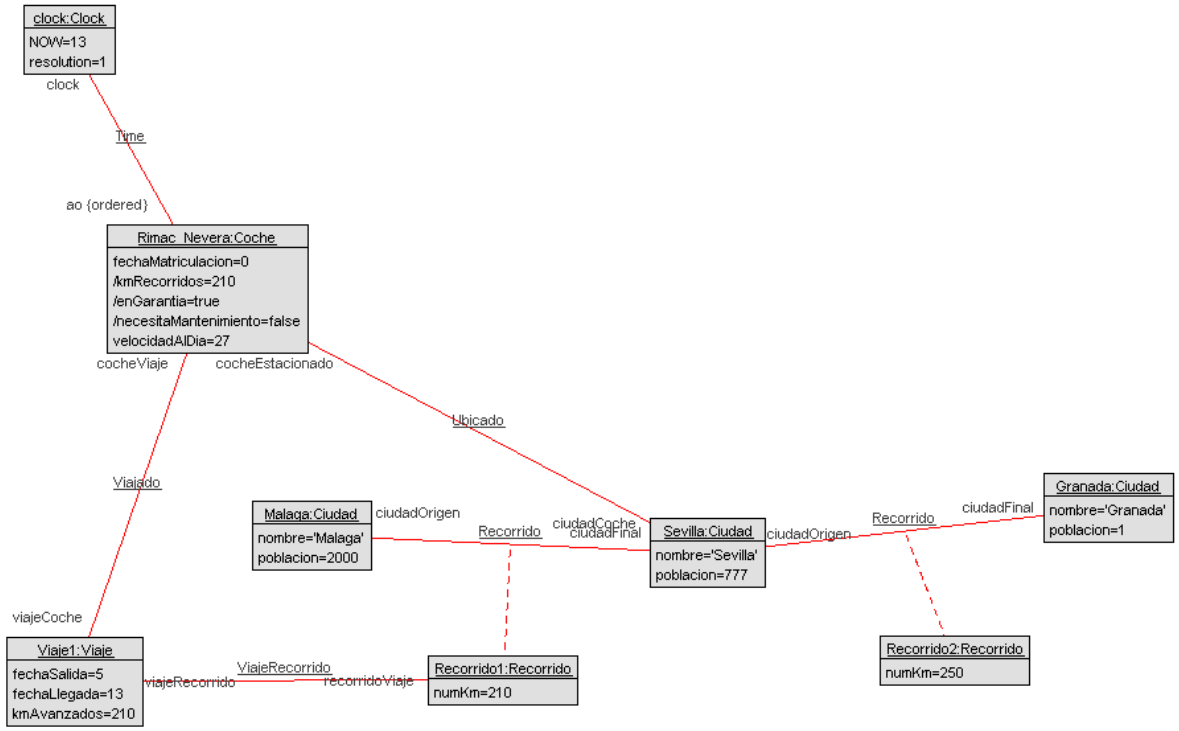
```
-- Continúa en Málaga hasta el día 5
!clock.run(5)
-- comienza un viaje haciendo el recorrido de Málaga a Sevilla
!Rimac_Nevera.comenzarViaje(Recorrido1)
-- Los días van pasando y el coche va avanzando hasta que llega a
Sevilla.
-- Necesitará 210/27 = 7.777 = 8 días
!clock.run(8)
-- El mismo día que llega a Sevilla, el coche comienza otro viaje
haciendo el recorrido entre Sevilla y Granada.
!Rimac_Nevera.comenzarViaje(Recorrido2)
--  Los días van pasando y el coche continúa realizando el viaje.
-- Necesitará 250/27 = 9.25 = 10 días
!clock.run(10)
-- Una vez llega a Granada, la simulación termina.
```

2.4.3.3. Diagramas de objetos del apartado C

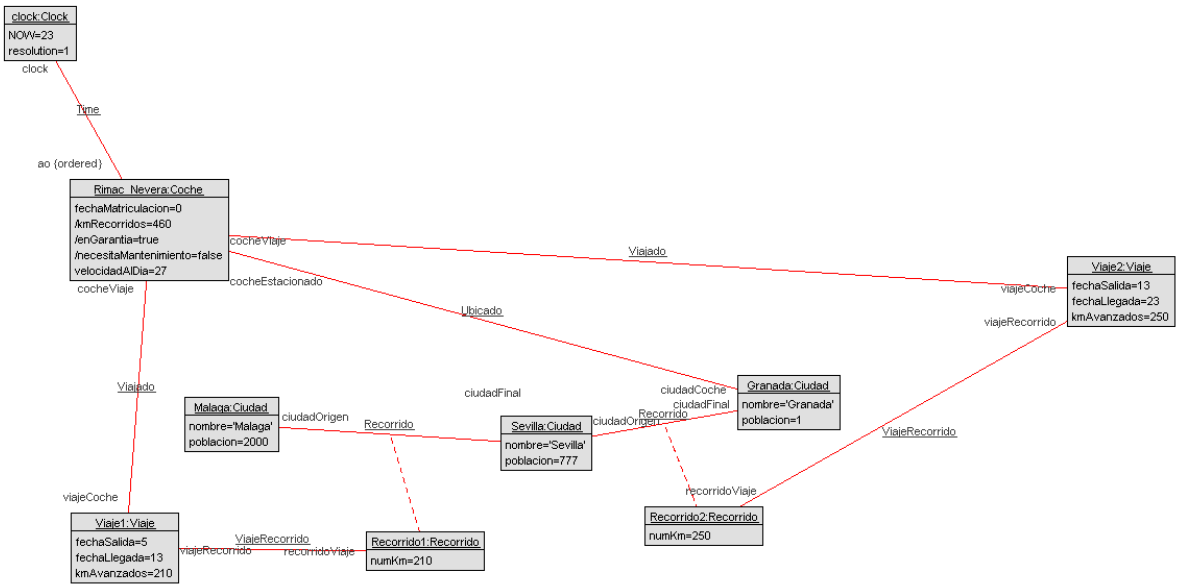
Tras ejecutar poco a poco estas operaciones sobre el soil, los diagramas en los instantes 0, cuando el coche llega a Sevilla y cuando el coche llega a Granada son:



5 Diagrama de objetos de apartado_c.soil en el instante 0



6 Diagrama de objetos de apartado_c.soil cuando el coche llega a Sevilla



7 Diagrama de objetos de apartado_c.soil cuando el coche llega a Granada

2.4.4. Código USE del apartado B

```
1. model sistemaCoches
2.
3. -------------- Enumerations ------------
4. enum TipoRevision { reparacion, mantenimiento}
5. -------------- End of enumerations ------------
6.
7. -------------- Classes ------------------
8.
9. class Clock
10.        attributes
11.          NOW : Integer init = 0 -- POSIX representation starting
   at 0
12.          resolution: Integer init = 1
13.        operations
14.          tick()
15.            begin
16.              -- Incremento el tiempo en dias
17.              self.NOW := self.NOW + self.resolution;
18.
19.              -- Notifico a todos los AO que avancen un día
20.              for o in self.ao do
21.                o.action()
22.              end;
23.            end
24.            post TimePasses: self.NOW = self.NOW@pre +
   self.resolution
```

```
25.
26.          run (n:Integer)
27.            begin
28.              -- Hago tick n veces
29.              for i in Sequence{1..n} do
30.                self.tick();
31.              end;
32.            end
33.
34.      end
35.
36.      abstract class ActiveObject -- real-time objects
37.        operations  -- para que Coche obligatoriamente implemente
   estas operaciones
38.          action() begin end
39.      end
40.
41.      class Coche < ActiveObject
42.        attributes
43.          fechaMatriculacion : Integer
44.
45.          -- esto es derive (apartado 13)
46.           kmRecorridos : Integer derive :
47.            self.viajeCoche->collect(v | v.kmAvanzados)->sum()
48.
49.          -- esto es derive (apartado 15) - SE USA CLOCK
50.          enGarantia : Boolean derive :
51.            ((self.clock.NOW - self.fechaMatriculacion) <= 4 * 100)
   or -- No han pasado más de 4 años desde la matriculacion
52.              -- Hay una revision en un taller oficial y aun le queda
   garantia
53.              self.revisionCoche->exists(r |
   r.tallerRevision.oclIsTypeOf(TallerOficial) and (self.clock.NOW -
   r.fechaFin <=
   r.tallerRevision.oclAsType(TallerOficial).tiempoGarantia))
54.
55.          -- esto es derive (apartado 16) - SE USA CLOCK
56.          necesitaMantenimiento : Boolean derive :
57.              -- Saco la fecha del último mantenimiento
58.              let fechaUltimoMantenimiento : Integer =
   (self.revisionCoche->select(r | r.tipoRevision =
   TipoRevision::mantenimiento)->sortedBy(r | r.fechaFin)-
   >last().fechaFin) in
59.
60.              ((self.clock.NOW - self.fechaMatriculacion) > 4 *
   100) and -- Han pasado más de 4 años desde la matriculacion
61.              ((self.revisionCoche->isEmpty()) or -- Si no tiene
   revisiones, le toca revisión
```

```
62.                 (self.clock.NOW - fechaUltimoMantenimiento
   >=  100)) -- Si las tiene, le toca si pasó 1 año desde la última de
   mantenimiento
63.
64.           -- indica cuantos km avanza al dia en los viajes
65.           velocidadAlDia : Integer
66.
67.         operations
68.           comenzarViaje(r : Recorrido)
69.             begin
70.               -- Crear un nuevo viaje con los valores iniciales
71.               declare nuevoViaje : Viaje;
72.               nuevoViaje := new Viaje;
73.               nuevoViaje.fechaSalida := self.clock.NOW;
74.               -- Asociar el nuevo viaje con el coche y el recorrido
75.               insert(self, nuevoViaje) into Viajado;
76.               insert(r, nuevoViaje) into ViajeRecorrido;
77.
78.               -- Al haberse comenzado un viaje, el coche ya no está
   en una ciudad, ¡está viajando!
79.               delete(self, self.ciudadCoche) from Ubicado;
80.             end
81.
82.             -- Un coche comienza un viaje desde la ciudad en la que
   se encuentra
83.             pre DesdeCiudadDondeSeEncuentra : (self.ciudadCoche =
   r.ciudadOrigen)
84.
85.             -- Ahora hay un viaje más, ese viaje tiene los
   atributos anteriormente instanciados,
86.             -- el coche ya no está ubicado en una ciudad y hay un
   viaje cuyo recorrido es el de esta operacion
87.             post ViajeCreado : ((self.viajeCoche->size() =
   self.viajeCoche@pre->size() + 1) and
88.                                 (self.viajeCoche -> exists(v |
   v.fechaLlegada.isUndefined() and v.fechaSalida = self.clock.NOW and
   v.kmAvanzados = 0))) and
89.                                 self.ciudadCoche.isUndefined() and
90.                                 self.viajeCoche->exists(v |
   v.recorridoViaje = r)
91.
92.           avanzar()
93.             begin
94.               -- Selecciona el viaje actual
95.               declare viajeActual : Viaje;
96.               viajeActual := self.viajeCoche->any(v |
   v.fechaLlegada.isUndefined());
97.
```

```
98.             viajeActual.kmAvanzados := viajeActual.kmAvanzados +
   self.velocidadAlDia;
99.
100.            -- Si el viaje se completa, establece la fecha de
   llegada
101.            if viajeActual.kmAvanzados >=
   viajeActual.recorridoViaje.numKm then
102.              viajeActual.kmAvanzados :=
   viajeActual.recorridoViaje.numKm;
103.              viajeActual.fechaLlegada := self.clock.NOW;
104.
105.              -- El coche se debe parar y crear la relación de
   Ubicado en la ciudad final
106.              insert(self,
   viajeActual.recorridoViaje.ciudadFinal) into Ubicado;
107.            end;
108.
109.          end
110.          post ViajeSigue_O_ViajeFinalizado :
111.              -- Postcondición 1: Si el viaje sigue en curso,
   kmAvanzados ha aumentado sin finalizar el viaje
112.              (self.viajeCoche->exists(v |
   v.fechaLlegada.isUndefined() and v.kmAvanzados = v.kmAvanzados@pre
   + self.velocidadAlDia) and
113.              self.ciudadCoche.isUndefined())
114.
115.              or
116.              -- Postcondición 2: Si el viaje se completa, el
   coche ha llegado a la ciudad final y tiene fecha de llegada
117.              (self.viajeCoche->exists(v |
   v.fechaLlegada.isDefined() and v.kmAvanzados =
   v.recorridoViaje.numKm and v.fechaLlegada = self.clock.NOW and
118.              self.ciudadCoche = v.recorridoViaje.ciudadFinal))
119.
120.        action()
121.          begin
122.            -- Simplemente tengo que llamar a avanzar, para si el
   coche está en un viaje, que avance (si no lo está no hará nada ya
   que no se cumplirá la precond)
123.            if (self.viajeCoche -> exists(v |
   v.fechaLlegada.isUndefined())) then -- El coche está realizando un
   viaje
124.              self.avanzar()
125.            end;
126.          end
127.
128.
129.    end
```

```
130.
131.     class Ciudad
132.       attributes
133.         nombre : String
134.         poblacion : Integer
135.     end
136.
137.     associationclass Recorrido between    -- CiudadOrigen y
   CiudadFinal, creo que deberemos realizar 2 caminos por cada 2
   ciudades
138.         Ciudad [0..*] role ciudadOrigen -- Esto es para facilitar
   el modelador
139.         Ciudad [0..*] role ciudadFinal
140.       attributes
141.         numKm : Integer
142.     end
143.
144.     class Viaje
145.       attributes
146.         fechaSalida : Integer
147.         fechaLlegada : Integer
148.
149.         -- El sistema también va a almacenar los kilómetros de
   los viajes.
150.         -- Cuando un viaje está en curso, este atributo indica
   los kilómetros que lleva realizado.
151.         kmAvanzados : Integer init : 0
152.     end
153.
154.     class Revision
155.       attributes
156.         fechaInicio : Integer
157.         fechaFin : Integer
158.         tipoRevision : TipoRevision
159.     end
160.
161.     class Taller
162.     end
163.
164.     class TallerOficial < Taller
165.       attributes
166.         tiempoGarantia : Integer
167.     end
168.
169.     -------------- End of classes ------------------
170.
171.     -------------- Relationships ------------------
172.     association Time between
```

```
173.       Clock [1] role clock
174.       ActiveObject [*] role ao ordered
175.     end
176.
177.     composition UbicacionTaller between
178.         Ciudad [1] role ciudadTaller
179.         Taller [0..*] role tallerCiudad
180.     end
181.
182.     association Ubicado between
183.         Coche [0..*] role cocheEstacionado
184.         Ciudad [0..1] role ciudadCoche
185.     end
186.
187.     association Revisado between
188.         Coche [1] role cocheRevision
189.         Revision [0..*] role revisionCoche
190.     end
191.
192.     association Realizada between
193.         Taller [1] role tallerRevision
194.         Revision [0..*] role revisionTaller
195.     end
196.
197.     association Viajado between
198.         Coche [1] role cocheViaje
199.         Viaje [0..*] role viajeCoche
200.     end
201.
202.     association ViajeRecorrido between
203.         Recorrido [1] role recorridoViaje
204.         Viaje [0..*] role viajeRecorrido
205.     end
206.
207.     -------------- End of relationships ------------------
208.
209.     ------------ Invariants ------------
210.
211.     constraints
212.     ------------------------
213.     -- El formato para las invariantes será el siguiente:
214.     -- context [CLASE]
215.       --- [ANOTACIONES ADICIONALES]
216.       -- [Número de apartado en enunciado]. [Frase explícita del
    enunciado]
217.       --- [ANOTACIONES ADICIONALES]
218.       --inv [nombre del invariante]
219.         -- [declaración del invariante]
```

```
220.
221.     -- Los invariantes implícitos se indicarán en [Número de
   apartado en enunciado] con una ?
222.     -- Si tienes dudas, hay un ejemplo justo abajo
223.     -------------------------
224.
225.     context Clock
226.       ---
227.       -- ?. Debe haber, como mucho, un solo reloj en el sistema.
228.       ---
229.       inv RelojEsUnico :
230.         Clock.allInstances()->size()=1
231.
232.     context Recorrido
233.       ---
234.       -- 1. Cada ciudad debe estar al menos a 5 kilómetros de
   distancia de otra.
235.       ---
236.       inv DistanciaEntreCiudadesAlMenosCinco :
237.         self.numKm >= 5
238.
239.     context Revision
240.       ---
241.       -- 4. Todas las revisiones deben tener lugar después de que
   el coche se matriculase
242.       ---
243.       inv RevisionDespuesDeMatriculacion :
244.         self.cocheRevision.fechaMatriculacion < self.fechaInicio
245.
246.       ---
247.       -- ?. Las fechas de una revisión deben ser válidas.
248.       -- Para ser válidas, la fecha de inicio de una revisión es
   anterior a la fecha de finalización y todas las fechas deben ser
   positivas
249.       -- Además, la fecha de inicio siempre debe estar definida
250.       ---
251.       inv FechasRevisionValidas :
252.         self.fechaInicio.isDefined() and self.fechaInicio >= 0
   and
253.         (self.fechaFin.isDefined() implies (self.fechaInicio <=
   self.fechaFin and self.fechaFin>=0))
254.
255.     context Ciudad
256.       ---
257.       -- 6. En cada ciudad habrá, a lo sumo, un taller oficial,
   pudiendo
258.       -- haber varios talleres no oficiales.
```

```
259.        --- se comprobará mirando si el taller es oficial, que solo
    exista uno
260.        inv UnSoloTallerOficial :
261.          self.tallerCiudad->select(t |
    t.oclIsTypeOf(TallerOficial))->size() <= 1
262.
263.      context Viaje
264.        ---
265.        -- 8. Si el coche está realizando actualmente algún viaje,
    dicho
266.        -- viaje únicamente tendrá fecha de salida, pero no de
    llegada.
267.        ---
268.        inv ViajeEnCurso :
269.          self.cocheViaje.ciudadCoche.isUndefined() implies
    self.fechaLlegada.isUndefined()
270.
271.        ---
272.        -- ?. Las fechas de un viaje deben ser válidas.
273.        -- Para ser válidas, la fecha de salida de un viaje es
    anterior a la fecha de llegada y todas las fechas deben ser
    positivas.
274.        -- Además, la fecha de salida siempre debe estar definida.
275.        -- También deben ser posteriores a la fecha de
    matriculación del vehículo que realiza el viaje.
276.        ---
277.        inv FechasViajeValidas :
278.          self.fechaSalida.isDefined() and self.fechaSalida >= 0
    and
279.          (self.fechaLlegada.isDefined() implies (self.fechaSalida
    <= self.fechaLlegada and self.fechaLlegada >= 0)) and
280.          self.cocheViaje.fechaMatriculacion <= self.fechaSalida
281.
282.      context Coche
283.        ---
284.        -- ?. La fecha de matriculación debe ser válida, es decir,
    debe ser positiva.
285.        ---
286.        inv FechaMatriculacionValida :
287.          self.fechaMatriculacion >= 0
288.
289.        ---
290.        -- 5. Como mucho, un coche se debe someter a una revisión,
    como máximo, en un momento dado.
291.        ---
292.        inv UnaRevisionALaVez :
293.          self.revisionCoche->select(r | r.fechaFin.isUndefined())-
    >size() <= 1
```

```
294.
295.        ---
296.        -- 7. Si un coche está siendo sometido a una revisión,
   entonces el coche debe encontrarse en
297.        -- la misma ciudad donde está el taller.
298.        ---
299.        inv CiudadCocheEnRevision_Igual_CiudadTaller :
300.          self.revisionCoche->isEmpty() or  -- no tiene revisiones
301.          self.revisionCoche->exists(r | r.fechaFin.isUndefined()
   implies self.ciudadCoche = r.tallerRevision.ciudadTaller) -- o si
   tiene, la revisión que está teniendo es en la misma ciudad
302.          -- si el exists no encuentra nada, da correcto (según
   ChatGPT)
303.
304.        ---
305.        -- 9. Un coche se encontrará en todo momento bien
   realizando un viaje determinado o bien en
306.        -- una ciudad.
307.        ---
308.        inv CocheEnViajeOCiudad :
309.          (self.viajeCoche->isEmpty() and
   self.ciudadCoche.isDefined()) or  -- si nunca ha hecho un viaje,
   está en una ciudad
310.          (self.viajeCoche->exists(v |
   v.fechaLlegada.isUndefined()) and self.ciudadCoche.isUndefined())
   or -- si está haciendo un viaje, no está en una ciudad
311.          (self.ciudadCoche.isDefined() and not self.viajeCoche-
   >exists(v | v.fechaLlegada.isUndefined())) -- si está en una
   ciudad, no está haciendo un viaje
312.
313.        ---
314.        -- 10. Si el coche ha completado al menos un viaje y no se
   encuentra viajando, entonces debe
315.        -- encontrarse en la ciudad a la que llegó en su último
   viaje.
316.        ---
317.        inv CocheTerminaEnCiudad :
318.          (self.viajeCoche->isEmpty()) or   -- Tiene que dar
   correcto si ha completado 0 viajes
319.          (not self.viajeCoche->exists(v |
   v.fechaLlegada.isUndefined()) implies -- si el coche no está en un
   viaje
320.          (self.ciudadCoche = self.viajeCoche->sortedBy(v |
   v.fechaLlegada)->last().recorridoViaje.ciudadFinal)) -- la ciudad
   en la que está ahora es la ciudad a la que ha llegado en el último
   viaje
321.
322.        ---
```

```
323.        -- 11. Dos viajes no pueden solaparse en el tiempo, es
    decir, un viaje debe ocurrir siempre
324.        -- después de otro, pudiendo la fecha de llegada de un
    viaje coincidir con la fecha de salida
325.        -- del siguiente.
326.        ---
327.        inv ViajesNoSeSolapan :
328.          let viajesOrdenados : OrderedSet(Viaje) =
    self.viajeCoche->sortedBy(v | v.fechaSalida) in -- Saco los viajes
    ordenados
329.              (self.viajeCoche -> size() >= 2) implies -- Si tengo
    más de 2
330.              (self.viajeCoche-> forAll(v1,v2 | viajesOrdenados-
    >indexOf(v1) =
331.                                          viajesOrdenados->indexOf(v2)
    - 1 implies ((v1.fechaLlegada <= v2.fechaSalida)))) -- basicamente,
    viaje[i].fechaLlegada < viaje[j].fechaSalida | i < j
332.        ---
333.        -- 12. Las ciudades de origen y destino de los viajes deben
    ser coherentes. Es decir, si un coche
334.        -- realiza un viaje desde la ciudad A hasta la ciudad B, el
    próximo viaje debe partir desde la ciudad B.
335.        ---
336.        inv CiudadesCoherentes :
337.          let viajesOrdenados : OrderedSet(Viaje) =
    self.viajeCoche->sortedBy(v | v.fechaSalida) in -- Saco los viajes
    ordenados
338.              (self.viajeCoche -> size() >= 2) implies -- Si tengo
    más de 2
339.              (self.viajeCoche-> forAll(v1,v2 | viajesOrdenados-
    >indexOf(v1) =
340.                                          viajesOrdenados->indexOf(v2)
    - 1 implies (v1.recorridoViaje.ciudadFinal =
    v2.recorridoViaje.ciudadOrigen))) -- si llegas a una ciudad, en el
    siguiente viaje partes desde ella
341.        ---
342.        -- 14. ESTO NO ES UN INVARIANTE, ES UNA ASUNCIÓN QUE SE
    HACE PARA FACILITAR EL MODELADO
343.        ---
344.        ---
345.        -- 15. ESTO NO ES UN INVARIANTE, ES EL DERIVE DE enGarantia
346.        ---
347.        ---
348.        -- 16. ESTO NO ES UN INVARIANTE, ES EL DERIVE DE
    necesitaMantenimiento
349.        ---
350.      ---------------- End of invariants -------------------
351.
```